The Lateralus Module System

Namespacing, visibility, package management, and the module import graph

Lateralus Language

bad-antics · March 2025 · Lateralus Language Research

ABSTRACT The Lateralus module system provides namespacing for definitions, visibility control for APIs, and a package manager interface for external dependencies. Each source file is a module; modules are organized into packages; packages are versioned and distributed via the Lateralus package registry. This paper describes the module declaration syntax, the import graph model, visibility rules, and the package manifest format. We pay special attention to the module system's interaction with the pipeline type system: a module can export pipeline stages as typed values, enabling modular composition of pipeline libraries.

1. Modules and Files

In Lateralus, every source file is a module. The module name is derived from the file path relative to the package root: a file at src/data/parser.lt in package my_app has module path my_app::data::parser.

A module may contain declarations in any order: the compiler resolves all names in a two-pass process that first scans top-level declarations and then resolves references. Forward references within a module are permitted.

```
// src/data/parser.lt
module my_app::data::parser

pub use crate::types::ParsedDoc

pub fn parse(input: &[u8]) -> Result<ParsedDoc, ParseError> {
    // ...
}

// Private helper — not exported
fn read_header(bytes: &[u8]) -> Header { ... }
```

2. Visibility: pub, pub(crate), and private

Lateralus has three visibility levels:

- **private** (default): the item is visible only within the declaring module.
- **pub(crate)**: the item is visible within the current package but not to external packages.
- **pub**: the item is visible to any importer, including external packages.

```
pub fn exported_to_everyone() { }
pub(crate) fn exported_within_package() { }
fn private_to_this_module() { }
```

The visibility system prevents accidental exposure of internal implementation details. A package's public API is exactly its set of pub-marked items at any visibility-boundary module path.

3. Importing Modules

The import keyword brings names from another module into scope:

```
// Absolute import
import std::data::Vec

// Multiple names from one module
import std::math::{ sin, cos, pi }
```

```
// Rename on import
import std::data::HashMap as Map

// Glob import (use sparingly)
import std::prelude::*
```

Imports are resolved at compile time. Cyclic imports are permitted as long as they do not create a cycle in the type definition graph (value cycles are resolved at link time, type cycles cause a compile error).

3.1 The Import Graph

The compiler builds an import graph before type-checking: a directed acyclic graph where nodes are modules and edges are import relationships. The topological order of the graph determines the compilation order: a module is compiled only after all its imports are compiled.

4. The Package Manifest

A package is a directory containing a Package.toml manifest and a src/ directory of Lateralus source files. The manifest declares the package name, version, and dependencies:

```
[package]
name    = "my_app"
version = "1.2.0"
edition = "2025"

[dependencies]
std          = { version = ">=1.0" }  # always implicit
http_client  = { version = "~0.8" }
json         = { version = "^2.1", features = ["streaming"] }

[dev-dependencies]
test_helpers = { version = "0.3" }
```

Version constraints follow SemVer: ^ allows any compatible version (same major), ~ allows patch-level changes only, and >= imposes a lower bound.

5. Pipeline Stages as Module Exports

One of the module system's distinctive features is the ability to export pipeline stage values as typed library items. A module can define a pipeline value and export it:

```
// Exporting a pipeline as a library value
pub let request_pipeline : Pipeline<RawBytes, HttpResponse> = pipe {
    |>  parse_http_request
    |?> authenticate
    |?> route_to_handler
    |>  serialize_response
}
```

External code imports and composes the pipeline:

```
import my_framework::request_pipeline

// Extend with additional middleware
let extended = logging_stage >> request_pipeline >> metrics_stage
```

This model enables pipeline library authors to ship composable stage values rather than just functions, giving users a richer vocabulary for building complex pipelines from trusted components.

6. Package Registry

The Lateralus package registry (pkg.lateralus.dev) hosts published packages. To publish a package:

```
ltl package build       # verify package, run tests
ltl package publish     # upload to registry
                         # requires API token from pkg.lateralus.dev
```

The registry enforces package name uniqueness per author, SemVer compliance, and a checksum manifest for every uploaded artifact. Package downloads are verified against the checksum before use.

7. Workspace Mode

Multiple packages that are developed together (a monorepo) are organized as a workspace. A workspace Workspace.toml at the repository root lists the packages:

```
[workspace]
members = [
    "packages/core",
    "packages/http",
    "packages/cli",
]
```

Workspace builds compile all members together, resolving cross-member imports as if they were internal modules. Path dependencies between workspace members do not require a registry round-trip.

8. Stability and Semver Checking

The Lateralus toolchain includes a semver checker that compares the public API of a new version against the previous version and determines the minimum version bump required (major, minor, patch). The check is based on the published API surface: added public items require a minor bump; removed or changed public items require a major bump; no public API changes allow a patch bump.

```
ltl package semver-check
# Comparing v1.2.0 API against published v1.1.3...
# Added: pub fn new_helper() [minor change: +1 minor version]
# Removed: none
# Changed: none
# Minimum required bump: 1.2.0 → 1.3.0
```

The semver checker integrates with the CI pipeline to block releases that would break the semver contract.